HPPS EXAM

Exam number 70

January 2023

Introduction

All benchmarks of the code have been made with 64 cores on MODI. All semantic tests have been made on my own 16 core workstation. All runs have been made using large data-sets, to make up for the constant overhead that multiple threads gives.
Note, that the matclib.c will only compile with a gcc that supports at least openMP 4.5, since that supports the array reduction clause, which I use.

0.1 Semantics

To make sure the semantics of the functions are correct, I have checked the calculations manually using "pen an paper", but also with CAS tools such as maple.

For parallelized functions with potential race conditions, I have made small programs to run them multiple times with multiple threads to make sure they give the same (correct) result each time.

0.1.1 Tests

To test the functions, I have made a test program called `semantic_test`, that tests all the function on some random matrices/vectors from the attached folder `test_files` against results found by using the CAS software Maple.
To run:

```
$ make all
$ ./semantic_test
```

0.2 Memory leaks

I have made sure that every time malloc has been used, there is a corresponding use of free. I have then also checked all functions using Valgrind, and made sure that they are all free from memory leaks. Even though Valgrind shows that there are "possibly lost"memory in some functions, this is most likely a false positive from Valgrind because of openMP.[1]

[1] See this article or this thread regarding this issue

1 `mul_vv()`

a

The function does have good temporal locality in the way, that the function is parallelized using static scheduling, and therefore even if every thread has their own private result variable (due to the reduction clause), the threads still fetches and updates the same variable multiple times, and therefore has good temporal locality.
The spatial locality is good in the function, since it traverses two flat arrays using a single loop, which means the elements it accesses are close to each other in memory.

b

I have parallelized the for-loop using the openMP clause

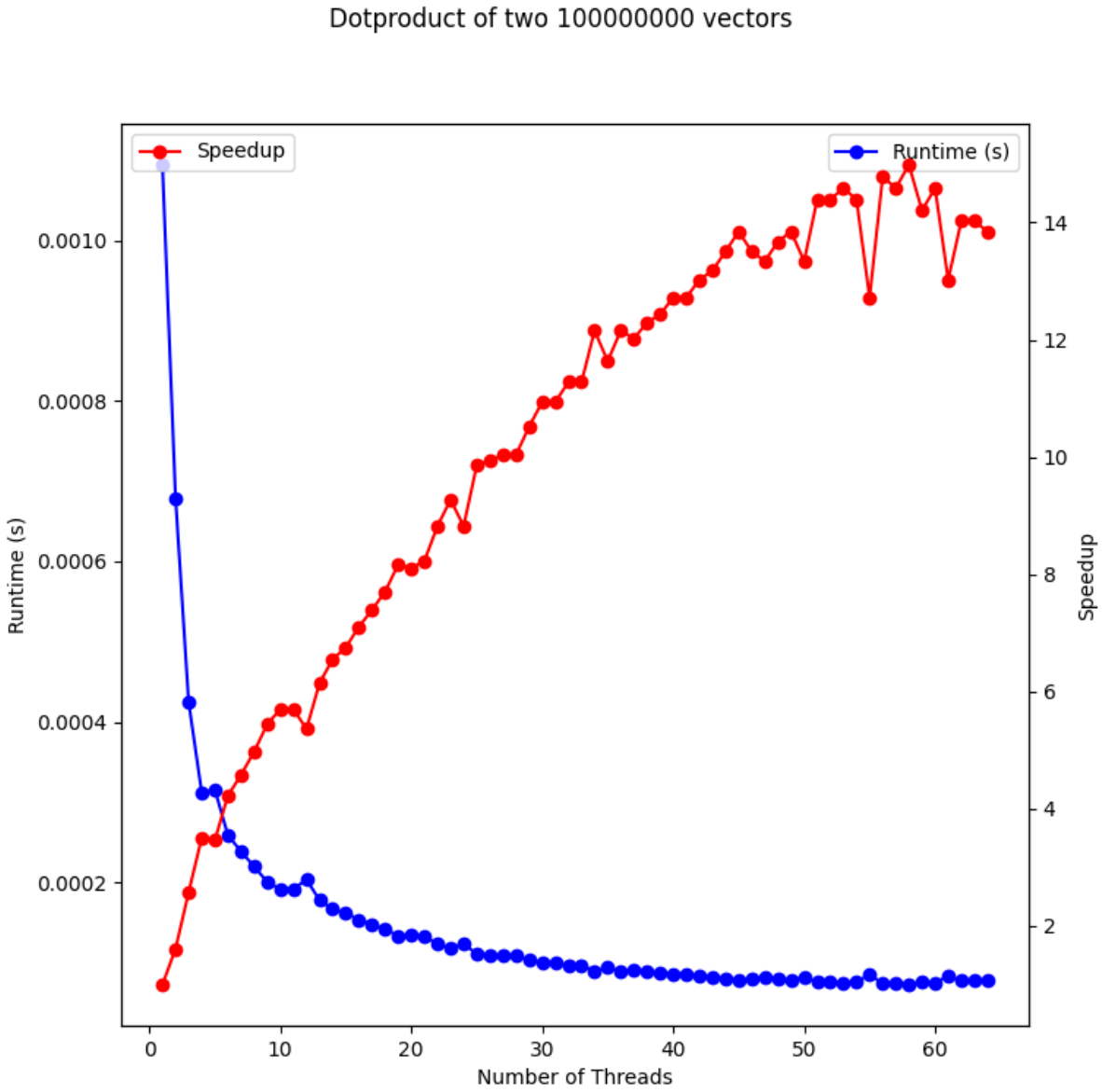
```
#pragma omp parallel for reduction(+: res)
```

Since the result variable addition results in a race condition for multiple threads, I included the reduction clause.
As discussed in subsection **a**, the function gains a lot from spatial locality, I have therefore chosen not to change the scheduling from the default (static), since static scheduling is best for spatial locality gained from traversing arrays. And since all computations are equal in size, there is no need for e.g. dynamic scheduling which is best when e.g. the last threads will have a substantial increased workload relative to other threads. [2]

c

To show how the strong scaling is, I have chosen to get the dotproduct two random vectors both with 1000000000 elements, since large arrays can better be utilized by multiple threads. We can see from the graph, see figure 1, that it has good strong scalability for the first couple of thread increases, but then converges at about a 14x speedup, and doesn't get much of an improvement for a thread increase.

[2] I also tested this for two 100000 vectors with 10 threads, and it was about 111 times faster with the default static scheduling instead of dynamic.

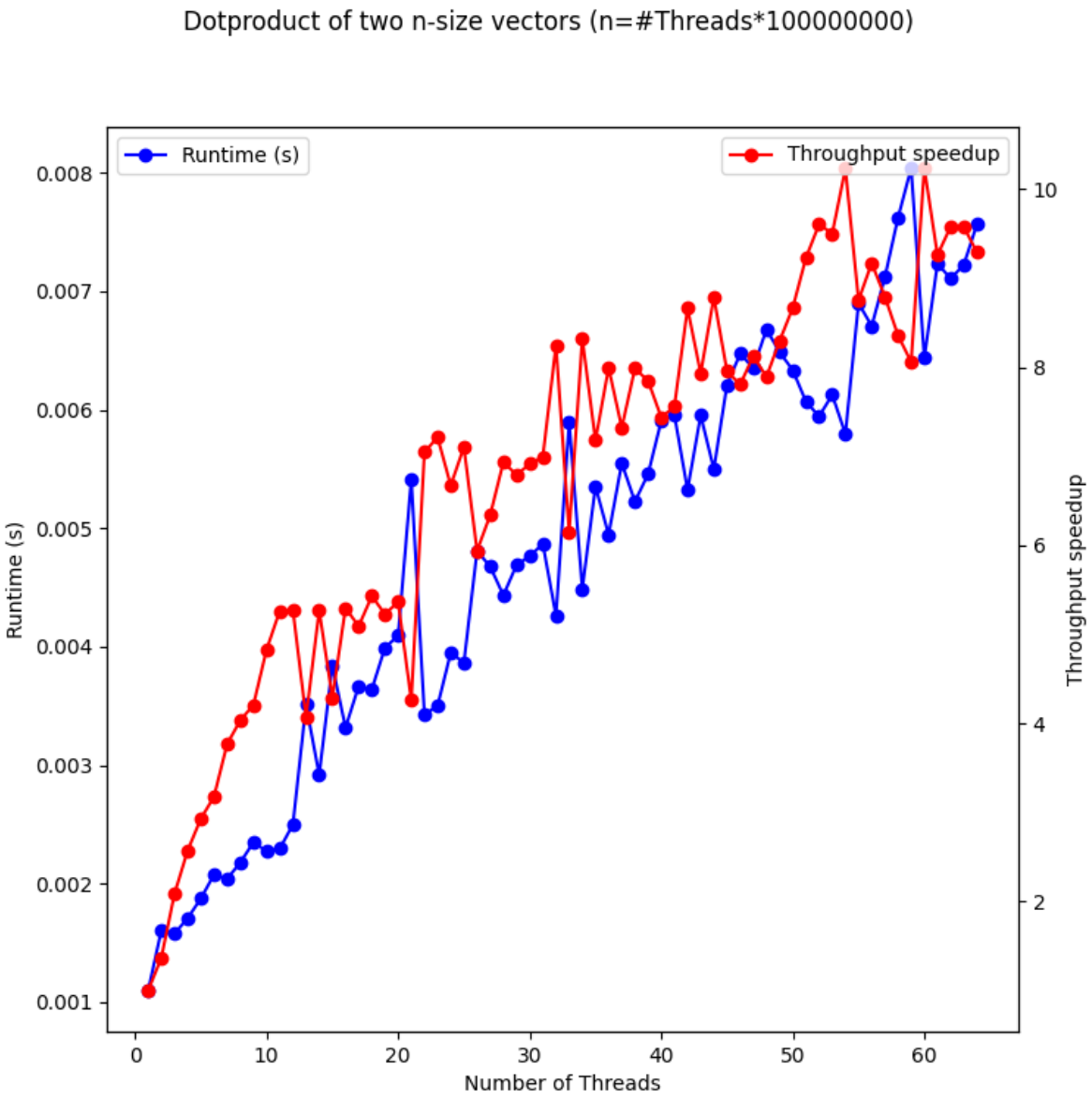


Figur 1: Average computation time for calculating the dotproduct of two vectors with different number of threads on MODI.

To test the weak scalability, I have made a similar test, but as number of threads increase, the size of the vectors (n) also increase like

$$n = \#Threads \cdot 1000000000$$

We can see from this graph, see figure 2, that it does have good weak scalability. The throughput speedup keeps increasing, but it does seem to be about a bit sub-linear/concave trending, so asymptotically it will have bad weak scaling.



Figur 2: Average computation time for calculating the dotproduct of two vectors with different number of threads and different sizes on MODI.

2 `mul_mv()`

a

Regarding the temporal locality of this function, it is good when only using one thread. But I am using multiple threads and therefore collapsing and using reduction, which makes every thread have their own copy of the result variable, and the temporal locality is therefore not that good.
The function however gains a lot from good spatial locality, since the matrix is in row major order, and the function traverses through the rows, and therefore accesses elements close to each other in memory. This is also only true, since the scheduling is static.

b

I have chosen to parallelize the main double for loop with the openMP clause

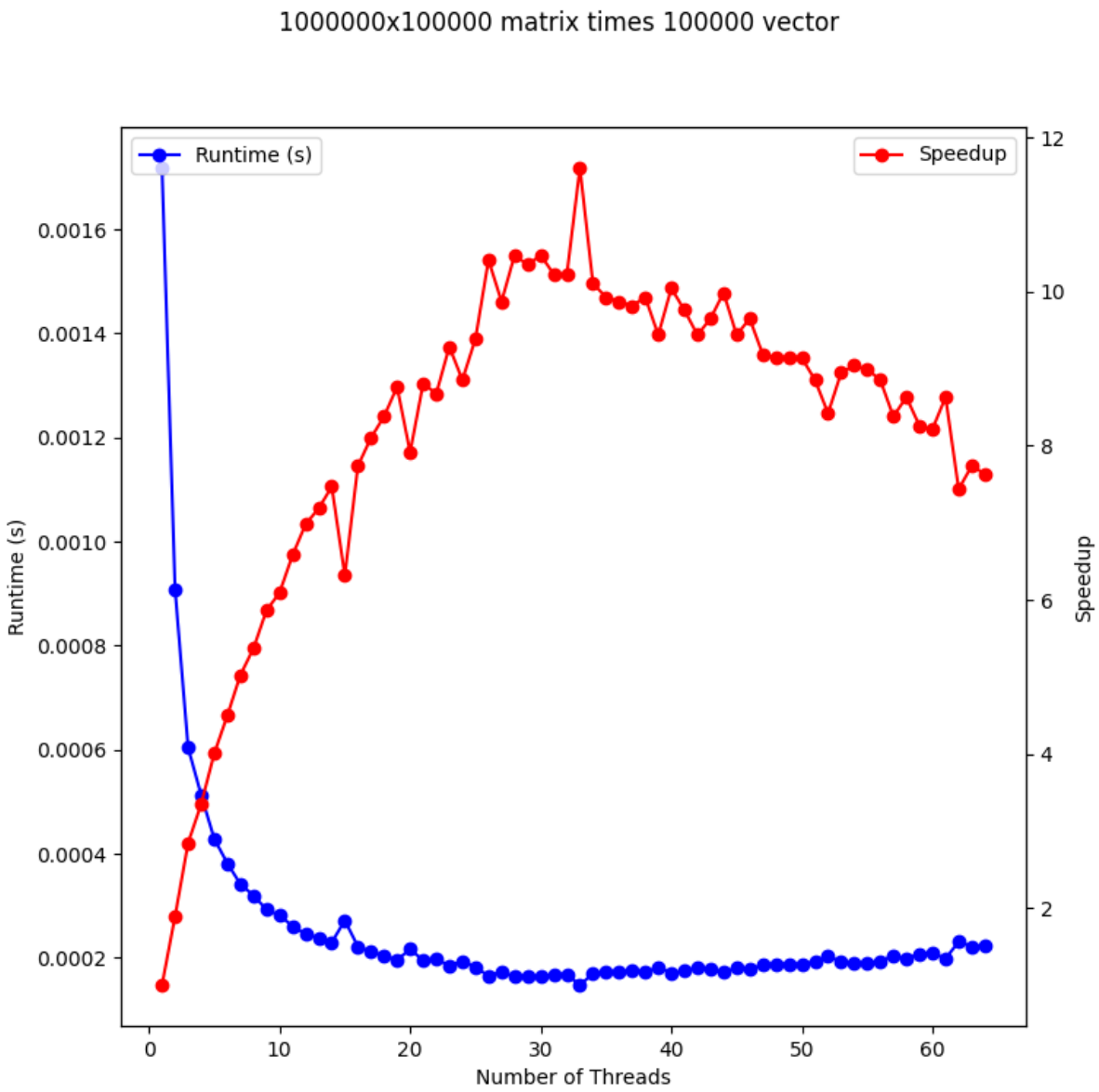
```
#pragma omp parallel for collapse(2) reduction(+:res[:A->n])
```

I collapse the two for loops, since they are perfectly nested, and then calculate the result array in multiple threads. And to avoid problems with the race condition when multiple threads want to update the same element, I have included a reduction for the result array. I again chose not to change the default scheduling, since the function, as discussed, gains a lot from spatial locality. [3]

c

To show how the strong scaling is, I have multiplied a 10000x100000 matrix on a 100000x1 vector. We can again see from the graph, see figure 3, that the function gains a lot from the first couple of thread increases, but then converges fast, and even decreases at about 30-40 threads. It therefore has very good strong scaling for the first 20-30 thread increases, but then has diminishing returns to scale asymptotically with regards to strong scaling.

[3] I again tested the theory with 10 threads and a 10000x10000 matrix times a 10000 vector, and as suggested the static scheduling was 73,5 times faster.

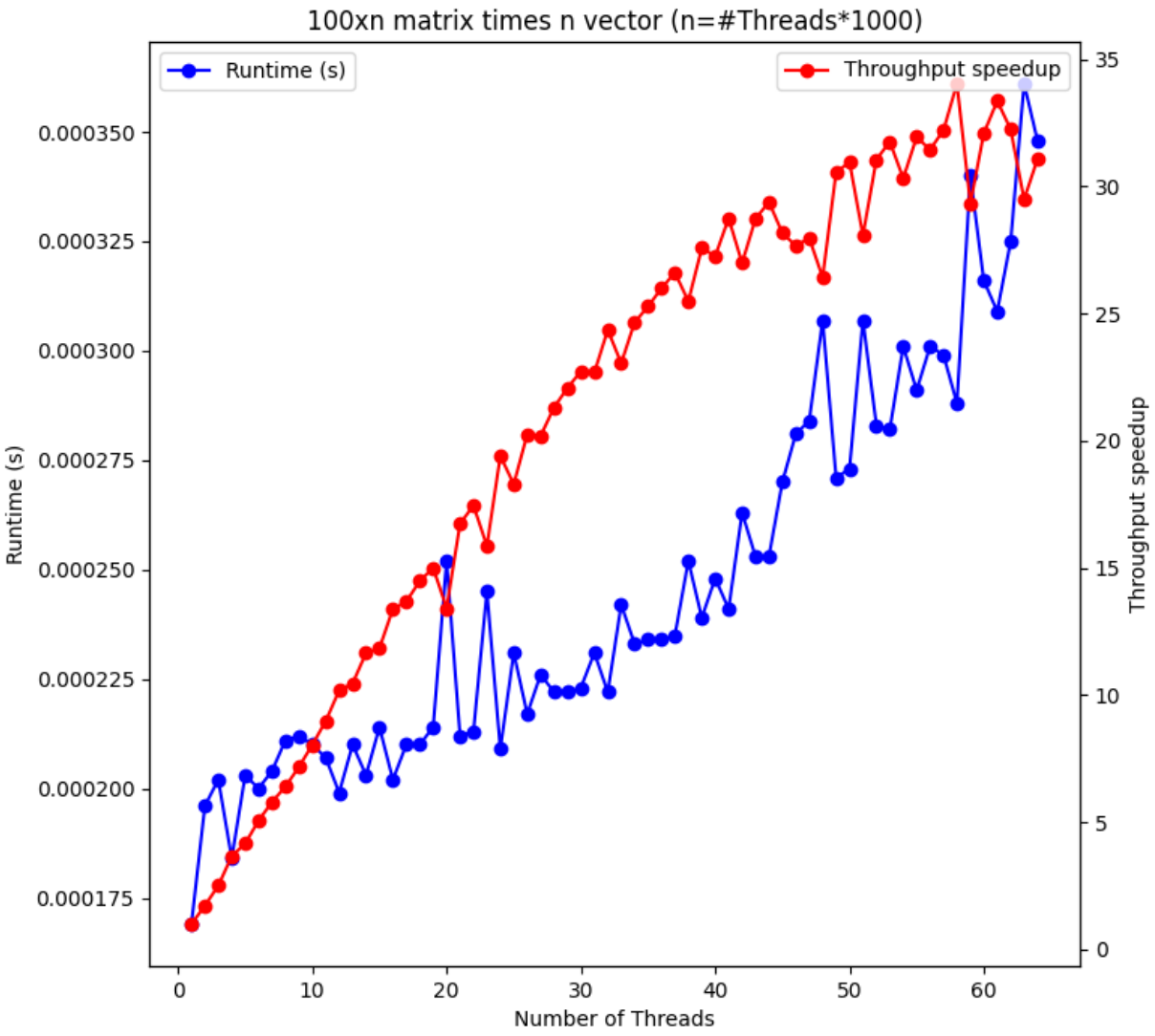


Figur 3: Average computation time for calculating a matrix times a vector with different number of threads on MODI

To test the weak scaling, I have made a similar test, but as number of threads increase, the size of the matrix and the size of the vector increase by

$$1000 \cdot \#Threads$$

In the graph, see figure 4, that it trends about linear, which in this case is good weak scaling. But it does seem like at about 50-60 threads it concaves downwards a bit, and therefore i would argue that the linear trend is not asymptotically. It therefore has good weak scaling for the first many increases, but again not asymptotically.



Figur 4: Average computation time for calculating a matrix times a vector (with sizes relative to thread count) with different number of threads on MODI

1 3 `mul_mTv()`

a

The temporal locality is by the same principle of `mul_mv`, since I use the reduction clause for the array, and therefore is only able to maybe benefit from temporal locality when the

threads make the final addition to the shared result array.
The spatial locality is quite good, since instead of traversing through the columns as you would do for a "normal"transpose multiplication, I traverse by the rows first, and then just jumping to the correct index of the result array. This makes the spatial locality of the matrix good, but worse of the result vector. But we would rather want to jump a lot in a small array of length n than in a matrix array with a size of nxm.[4]

b

I have chosen to parralalize the main double for loop with the following openMP clause:

```
#pragma omp parallel for collapse(2) reduction(+:res[:A->m])
```

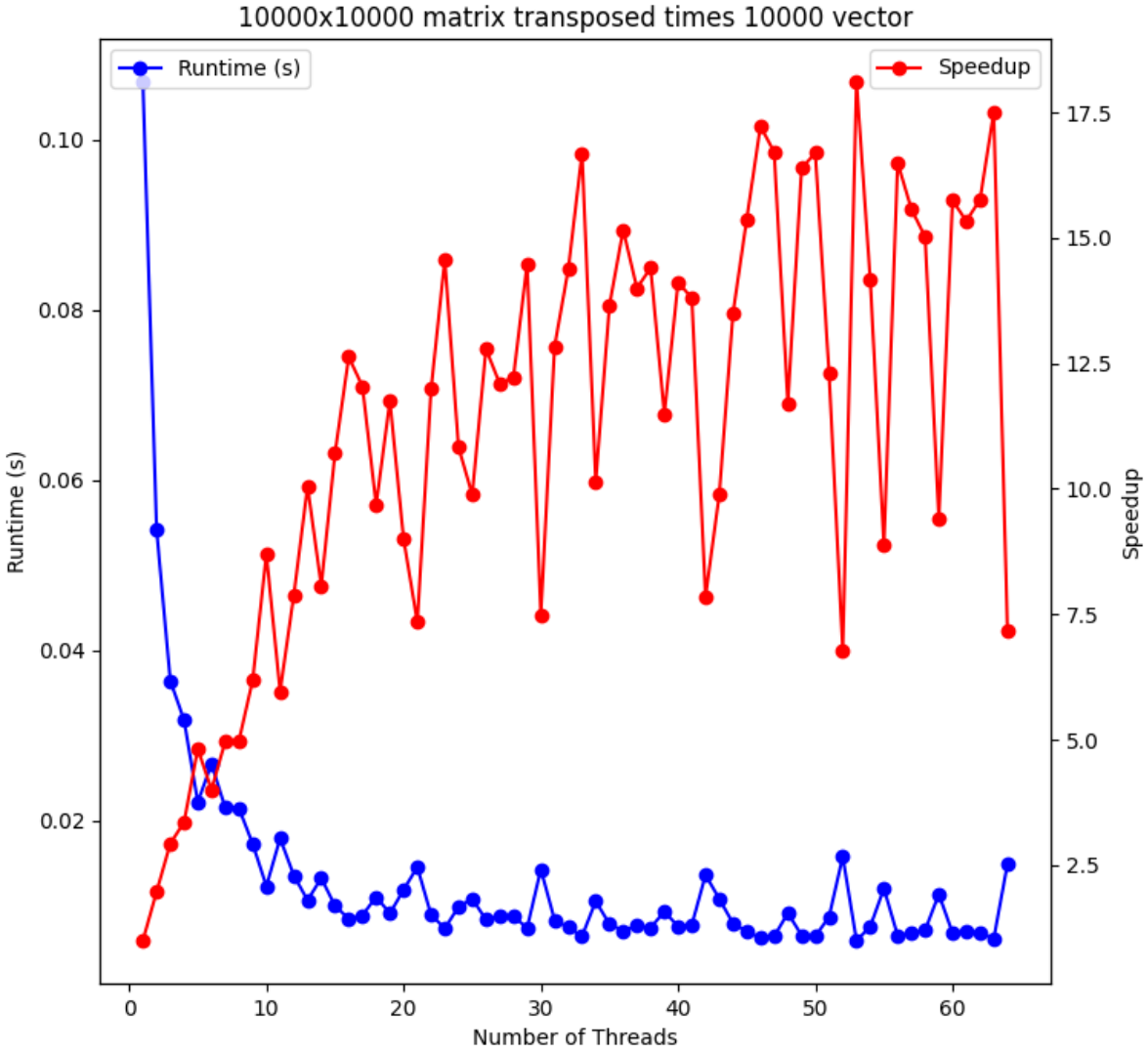
Collapsing the two for-loops since they are independent, and then making a reduction to handle for the race condition that the addition/update of the result vector makes. And again no change in scheduling, since same logic as earlier can be applied.[5]

c

To show the strong scaling of the function, I have again chosen to multiply a 10000x1000 matrix with a 10000 vector. On the graph, see figure 5, we can again clearly see, that it benefits a lot from the first couple of thread increases, but the again converges. Even though the data points are a bit unstable, it converges at about 15-17 speedup without regarding the outliers.

[4] I also tested this theory using both methods of traversing, using a 1000x1000 matrix times a 1000 vector, and as suggested the mean computation time for 1000 runs was about 1.7 times faster.

[5] But I of course tested it, and as suggested the static scheduling was about 71.8 times faster than dynamic for a 10000x10000 matrix with a 10000 vector.

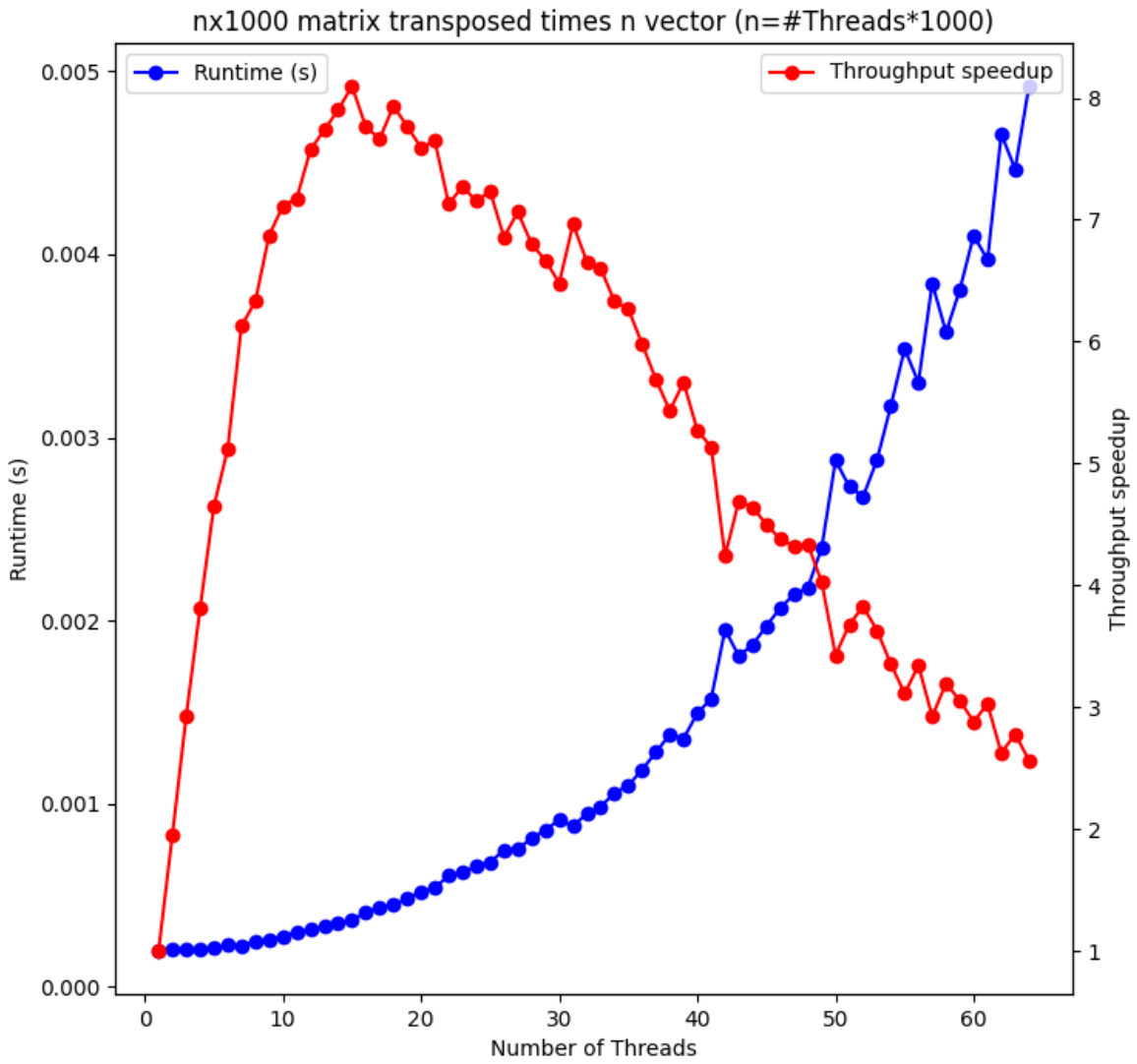


Figur 5: Average computation time (1000 runs) for for calculating a transposed matrix times a vector with different number of threads on MODI

To show the weak scaling, I have chosen to multiply a transposed matrix with nx1000 with a n vector, where

$$n = \#Threads \cdot 1000$$

On the graph, see figure 6, that the graph rises exponentially when the data set gradually increases in size, and the throughput speedup decreases as well at around 15 threads. Which means the function has really good weak scaling for the first 15 threads, but it has bad asymptotically weak scaling.



Figur 6: Average computation time for calculating a transposed matrix times a vector (with sizes relative to thread count) with different number of threads on MODI

4 `mul_spmv()`

a

The function uses the same variables like nnz and multiple arrays many times, and because of static scheduling the thread may gain some temporal locality from this, since the thread

uses them multiple times, and it is therefore likely that the variables/arrays are already in the cache of the CPU. The spatial locality is also good, since all the arrays used are flat, and the two for loops both traverse the arrays in a sequential order. The only array where the access might "jump"around is when accessing `x`'s elements at `v_col[l]`, and therefore maybe affect the spatial locality, if it "jumps around"too much, but wont necessarily affect it much.

b

I have chosen to parralalize the main loop using the openMP clause

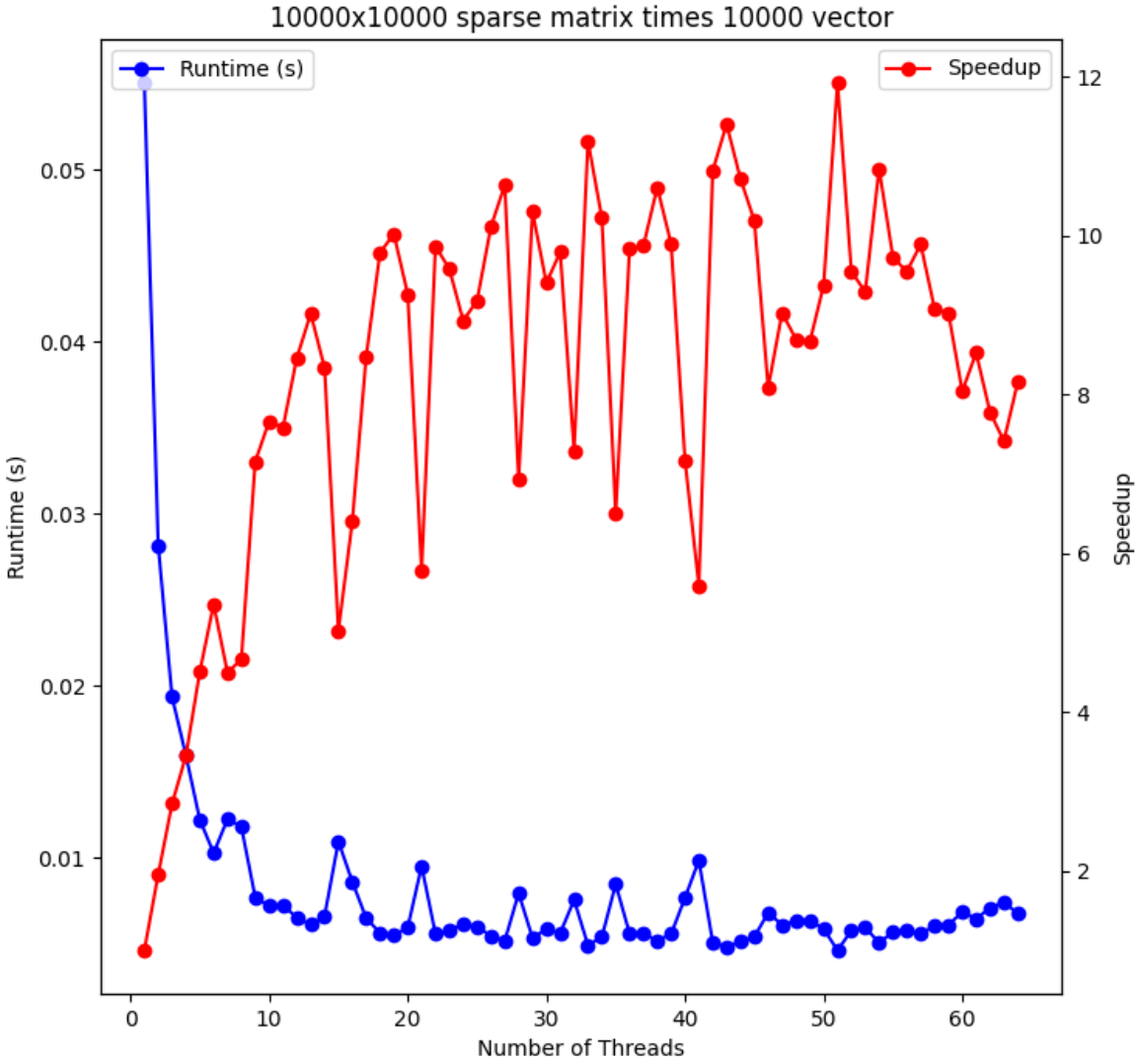
```
#pragma omp parallel for
```

Without any special clauses, since there is no race condition or other special cases. The update of the elements in the result array, `y`, might look like a race condition, but since all threads will have an unique value of i, they will never access the same element, and therefore no reduction clause is needed.
One might try to parralalize the nested inner for-loop, but that would result in too many threads, and lead to too much context switching and therefore making it slower. For example if I run the outer loop with 8 threads, all 8 will then create l threads again resulting in $l \cdot 8$ threads active, which is not good for performance on most cpu's.[6]

c

To test the strong scaling of the function, I have multiplied a 10000x10000 sparse matrix on a 10000 vector, and then increased the number of threads and measuring average time for each computation. On the graph, see figure 7, we see that the time decreases a lot for the first thread increases, but then converges as well as the speedup. We will therefore get about the same result if we run with 30, 40, or 50 cores. And it also seems like it maybe even slows down a bit at around 50 cores. It therefore as the other functions have great strong scaling in the beginning of thread increases, but then bad asymptotically.

[6] I also tested this theory with 8 threads (total of 16 cores available) and a 1000x1000 matrix times a 1000 vector for 1000 runs, and as suggested the one with the inner loop parralalized was on average 5.9 times slower.



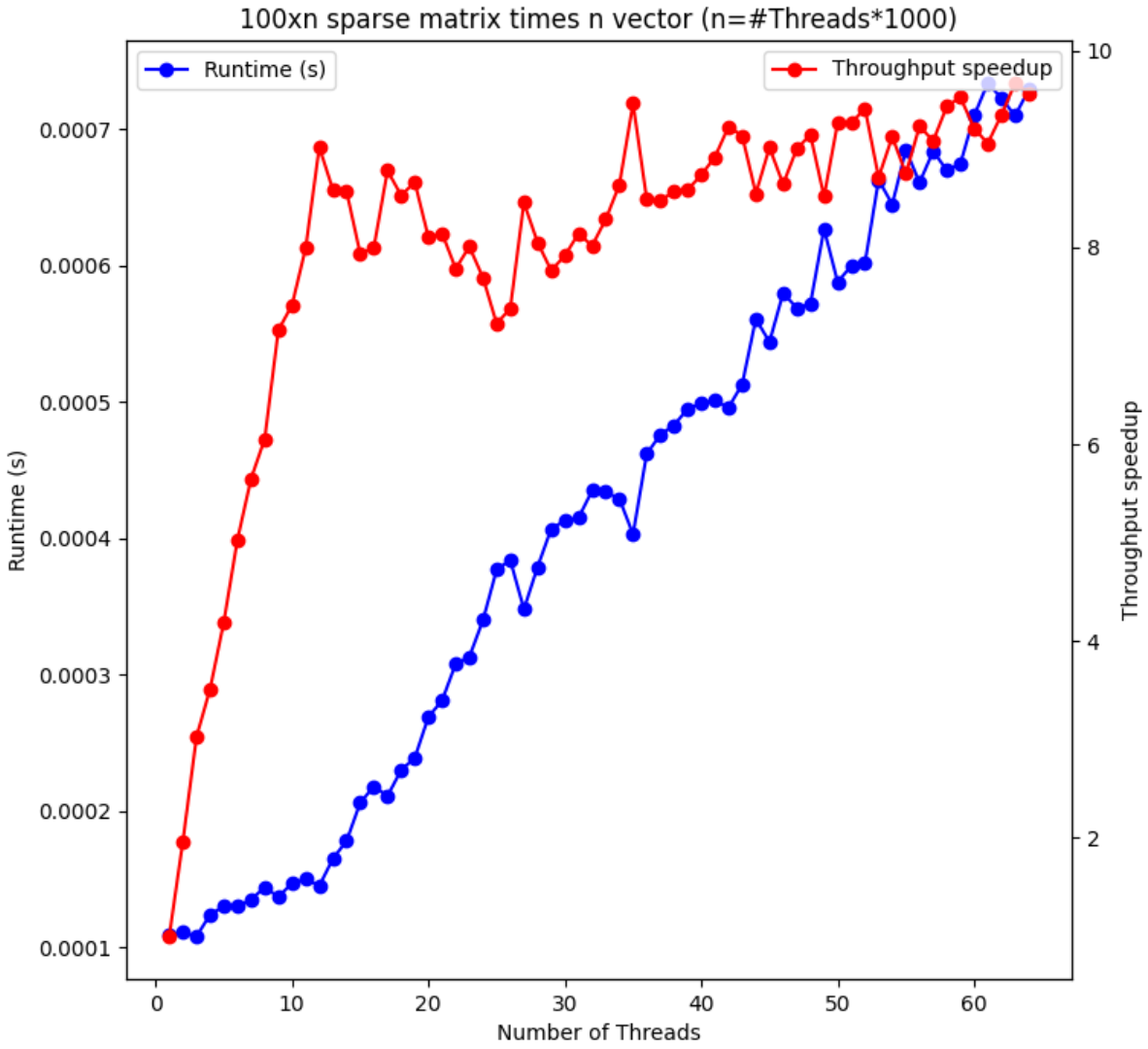
Figur 7: Average computation time for calculating a sparse matrix times a vector with different number of threads on MODI

To show the weak scaling, I have chosen to multiply a sparse matrix with 100xn size with a n vector, where

$$n = \#Threads \cdot 1000$$

On the graph, see figure 8, the throughput speedup seems to be about logarithmic trending. It first increases a lot but the quickly converges between 9-10x speedup. Which means that we will almost always see an increase in throughput when increasing number of threads, but with eventually large diminishing returns to scale. And it therefore has a bad asympto-

tically weak scaling.



Figur 8: Average computation time for calculating a sparse matrix times a vector (with sizes relative to thread count) with different number of threads on MODI

5 mul_spmTv()

a

Same logic as the `spmv` function, since we again use static scheduling, we will gain some temporal locality since each thread uses the same variables multiple times. The spatial locality is as well good for the most part, since all arrays, except `y`, is traversed in sequential order. The `y` array can, again with regards to the threads, get accesses in a non-sequeanital order, and therefore "jump around", which will result in not too good spatial locality.

b

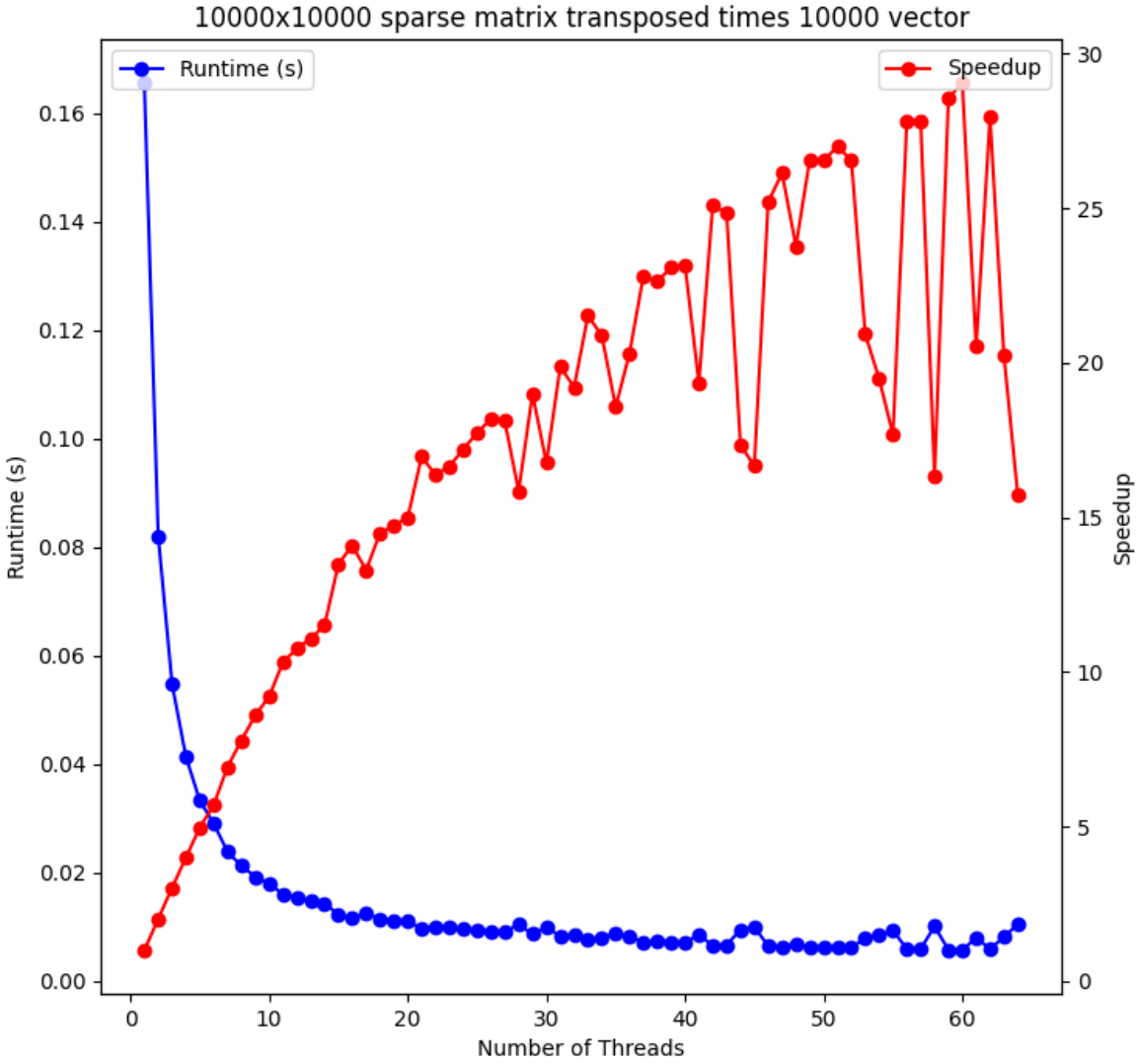
To parralalize the function, I have used the following openMP clause

```
#pragma omp parallel for reduction(+:y[:A->m])
```

As the same logic from `mul-spmv`, I have chosen only to parralalize the outer for-loop. This time I use a reduction clause, since the y array can be updated at the same element by multiple threads, and therefore makes a race condition.

c

To test the strong scaling, I have multiplied a 10000x10000 sparse matrix transposed on a 10000 vector. On the graph, see figure 9 it can again clearly be seen that the function gains a lot of speedup in the beginning of thread increases but then decreases. It trends about logarithmic in speedup, and therefore has diminishing returns to scale. It has good strong scalability for the first couple of thread increases, but not asymptotically.

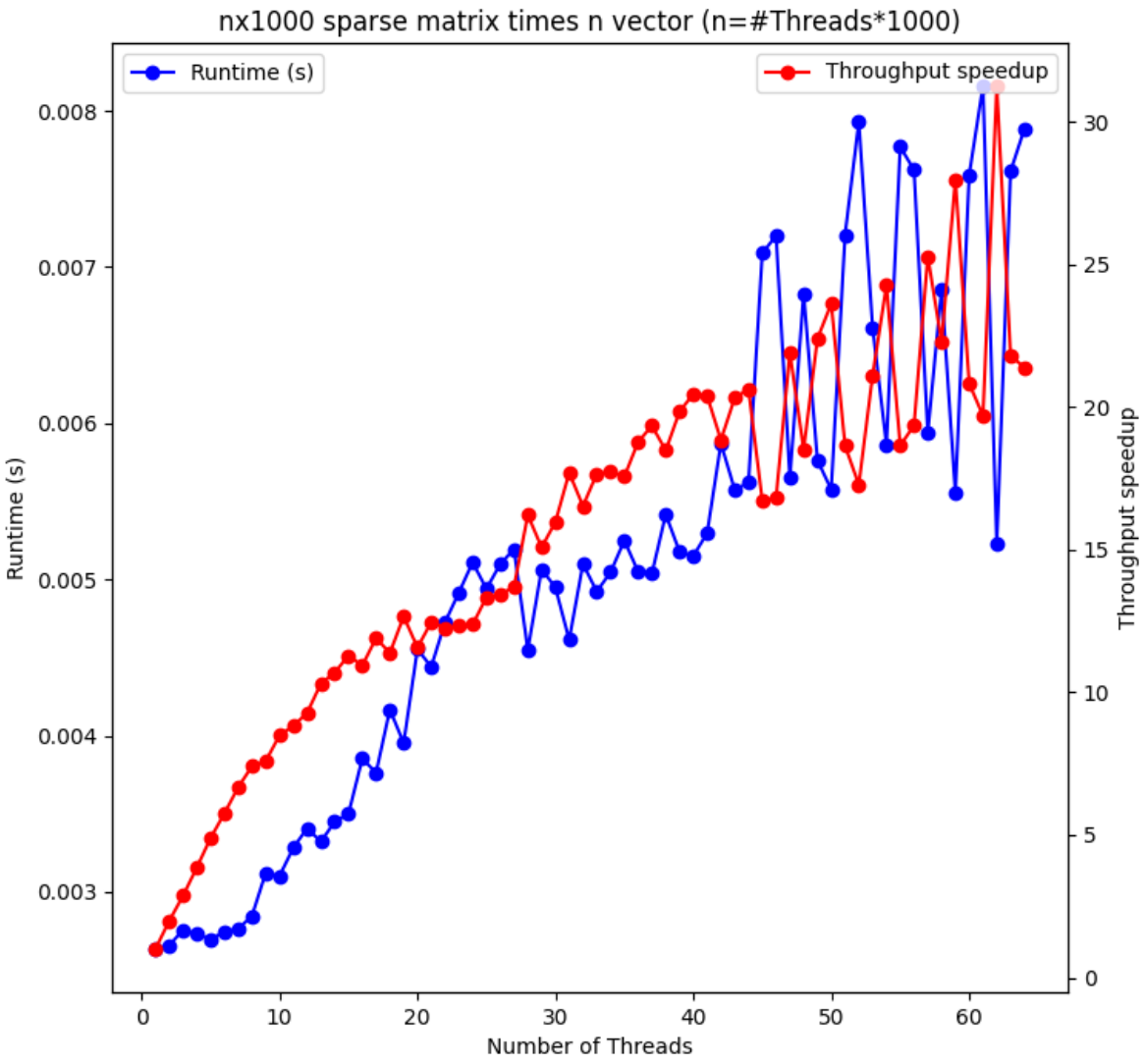


Figur 9: Average computation time for calculating a transposed sparse matrix times a vector with different number of threads on MODI

To show the weak scaling, I have chosen to multiply a transposed matrix with nx1000 with a n vector, where

$$n = \#Threads \cdot 1000$$

On the graph, see figure 10, the throughput speedup scales logarithmic, and therefore will still gain from thread increases when increasing workload, but not linear. And we will therefore have to increase number of threads more than the workload (asymptotically). But it is still really good, we see that it converges at about 25x and maybe even up to around 30x throughput speedup.



Figur 10: Average computation time for calculating a transposed sparse matrix times a vector (with sizes relative to thread count) with different number of threads on MODI